Learning Continuous Control Agents with Evolutionary Algorithms

Author: `Andrew Malta`

Introduction

In this project, I explored the application of evolutionary algorithms to learn policies for continuous control agents in the OpenAI Gym. The OpenAI Gym, while traditionally used to measure the performance of reinforcement learning approaches on toy environments from Atari games to motion tasks, provided me with an ideal framework to test the performance of evolutionary algorithms on learning deterministic policies for agents. I first got interested in this topic when I read OpenAI's paper, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning", which experimentally demonstrated that it is possible to rival the performance of reinforcement learning in many tasks while gaining the desirable property of highly parallelizable training procedures. This report along with an excellent blog post about applying these techniques to the OpenAI gym in particular, both referenced at the bottom, encouraged me to try my hand at some of the hardest control tasks offered in the OpenAI gym.

Environments

In choosing an environment to experiment with, I first looked for a task that would be particularly amenable to reinforcement learning, as I wanted to see that a typical reinforcment learning task could be solved through the use of an evolutionary algorithm. I settled on the Bipedal walking task in the OpenAI gym as it seemed to meet these criteria and due to the fact that it had both a normal and a hadcore version, which I thought I might be able to exploit in the training process. Very simply, the goal of the task is to have your agent navigate itself across the terrain in front of it and reach the goal on the other end. The reward function penalizes stumbling, defined to be when the head hits the ground, and the use of excess motor torque. A good solution is robust to the terrain, but does not use too much motor torque to maintain this robustness.

The input to the model is a stream of 24 dimensional vectors each of which includes hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, leg contacts with the ground, and 10 lidar inputs that encode objects in the vicinity. The output is a 4 dimensional vector representing the amount of torque to apply to each of the 4 joints on the agent.

The normal version of the environment has small random variation in the terrain, but all in all it is pretty flat.

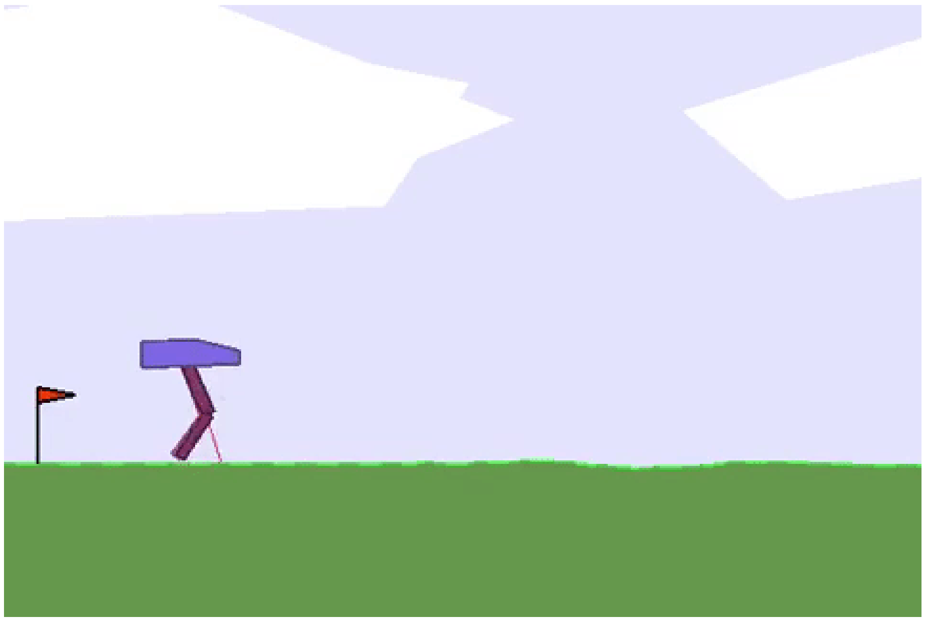

Figure 1: Alt Text

The hardcore of the environment, on the other hand, has pits, ramps, and obstructions of various sizes that the agent has to navigate. Due to this terrain, it is much easier for the agent to get stuck in local minima, as it can learn how to navigate some of the terrain but not the rest.

The hardcore task poses a major leap in difficulty as the terrain varies much more significantly for each run of the environment than the normal version of the environment does. This forces you to learn a robust policy to work in any configuration of these ramps, pits, and obstructions rather than learn a particular motor sequence that works for a particular environment. This was not a major issue in the normal bipedal walking environment as the terrain variation was minimal, allowing the evolutionary algorithm to simply learn a motor sequence that propelled itself forward while keeping its balance.

Model

For this project, I decided to roll my own small neural network framework as I wanted to have direct access to the parameterization of the weights and biases of the network through a single flat vector. This, in my mind, was the easiest way to interface the evoltionary algorithms with my model as most libraries make you go out of your way to do something other than the classic back propagation with some optimizer. I wanted to keep the model simple because I wanted to

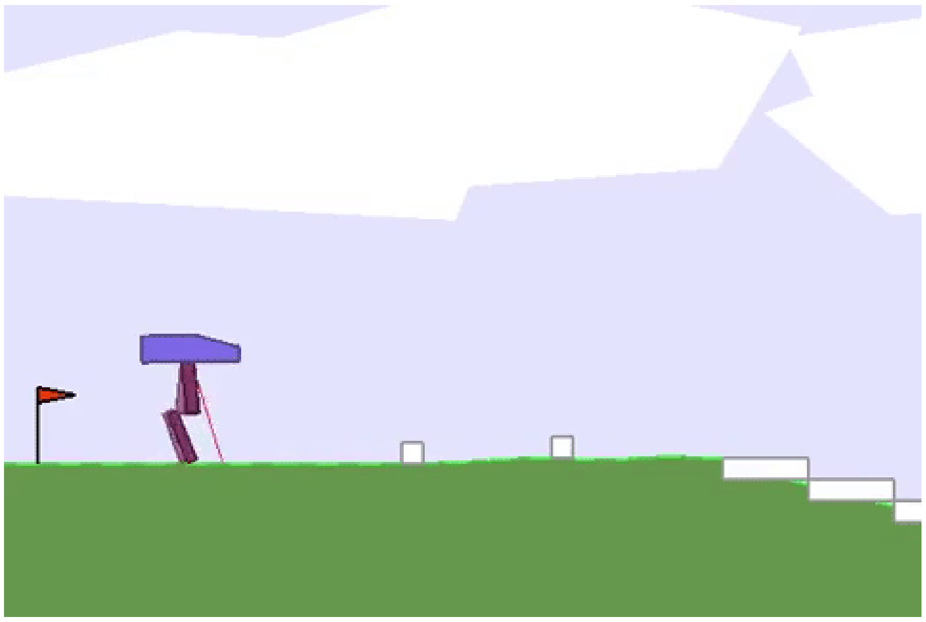

Figure 2: Alt Text

try keep the focus on the optimzation procedure, and I wanted the optimzation routine to run quickly.

The model I ended up choosing was a feed-forward nerual network with 2 hidden layers and hypberbolic tangent activation, which ended up totaling a parameter vector of length 2804 when you account for the biases. This number of parameters, of course, was a main concern when I was choosing the size of my model, as I wanted to ensure that the neural network was able to represent the function I was trying to approximate. That being said, I wanted to keep the parameter vector small enough for performance reasons, as I wanted to test the performance of the CMA-ES optimization algorithm, which scales poorly with the size of the parameter vector. The code for this simple neural network model is listed below:

```
import numpy as np

class Net():
  def __init__(self, weights):
    # an affine operation: y = Wx + b
    self.dimensions = [(24, 40), (40, 40), (40, 4)]
    self.layers = []
    self.biases = []
    if not weights is None:
      tmp = 0
      for d in self.dimensions:
        length = np.prod(d)
        self.layers.append(np.reshape(weights[tmp: tmp + length], d))
        tmp += length
        self.biases.append(np.reshape(weights[tmp: tmp + d[-1]], (1, d[-1])))
        tmp += d[-1]

  def set_model_params(self, weights):
    self.layers = []
    self.biases = []

    tmp = 0
    for d in self.dimensions:
      length = np.prod(d)
      self.layers.append(np.reshape(weights[tmp: tmp + length], d))
      tmp += length
      self.biases.append(np.reshape(weights[tmp: tmp + d[-1]], (1, d[-1])))
      tmp += d[-1]

  def forward(self, x):
    working_tensor = x
    for i in xrange(len(self.layers)):
      affine = np.dot(working_tensor, self.layers[i]) + self.biases[i]
      working_tensor = np.tanh(affine)
    return working_tensor[0]

  def num_flat_features(self):
    ret = 0
    for d in self.dimensions:
      ret += np.prod(d)
      ret += d[-1]
    return ret
```

Optimization

I played around with a number of evolutionary algorithms, some of which do not have a canonical name and other that certainly do. When I was performing experiments on some other environments that I will not include in the report, I was still not using the MPI interface I adapted from ESTool. I experimented with some of the basic strategies such as one that I ended up reffering to as WinnerES which was simply

WinnerES

$\theta_0 = \vec{0}$ for some number of generations: $\epsilon_{1,\ldots,n} \sim N(\sigma, I)$ $j = \text{argmax}_i f(\theta_t + \epsilon_i)$ $\theta_{t+1} = \theta_t + \epsilon_j$

This is a basic template for a evolutionary algorithm with a simplified update step of just choosing the best child from the generation and using it as the starting point of the next generation. This is not great for a number of reasons, but it could work for some simple tasks as I saw. In other words this is really just a randomized sampling of directions on the objective function to approximate the gradient at your current parameter vector. The larger your generation size, n, the better your approximation of the gradient gets, but the longer the computation takes per step.

The next algorithm I implemented and tested on some toy examples was the non-parallelized version of OpenAI's evolutionary strategy, which is described in algorithm 1 in the paper in the references. In my code I reference it as WeightedAverageES, and the updates are as follows:

OpenES(Single Threaded)

$\theta 0 = \vec{0}$

for some number of generations: $\epsilon 1, ..., n\ N(\sigma, I)$ $f_i = f(\theta_t + \epsilon_i)$ $\theta_{t+1} = \theta_t + \frac{\alpha}{n\sigma} \sum_i^n f_i * \epsilon_i$

Training on Grace

To exploit the parallel nature of the problem, I applied for and used the high performance computing cluster, Grace. This allowed for a dramatic speedup of the training procedure of the policy, and eventually led to the solutions that we will list below to the selected environments. I adapted the parallel training procedure used by ESTool, the open source project written by the author of the blogpost I referenced. This code used MPI as a way to communicate between multiple processes each running playouts of the environment simulation. This parallel playout strucutre, which follows the procedure that OpenAI used in their paper, allows us to test many different random mutations of our current best performing paraments while only having to communicate the random seed used to generate the mutation noise. This, of course, dramatically speeds up the amount of time each generation takes to complete. I used the following run script on Grace

```
#!/bin/bash
#SBATCH --partition=scavenge
#SBATCH --job-name=bipedal_walker_hardcore_es
#SBATCH --ntasks=64
```

```
#SBATCH --ntasks-per-node=1
#SBATCH --mem-per-cpu=6000
#SBATCH --time=12:00:00
#SBATCH --mail-type=ALL
#SBATCH --mail-user=email

source activate amth
pip install mpi4py
python train.py -e 16 -n 64 -t 4 --start_file <model_start_file>
```

Results

The results of the experiment were resoundingly positive. I was skeptical of the methods being able to work on the bipedal walking environmen

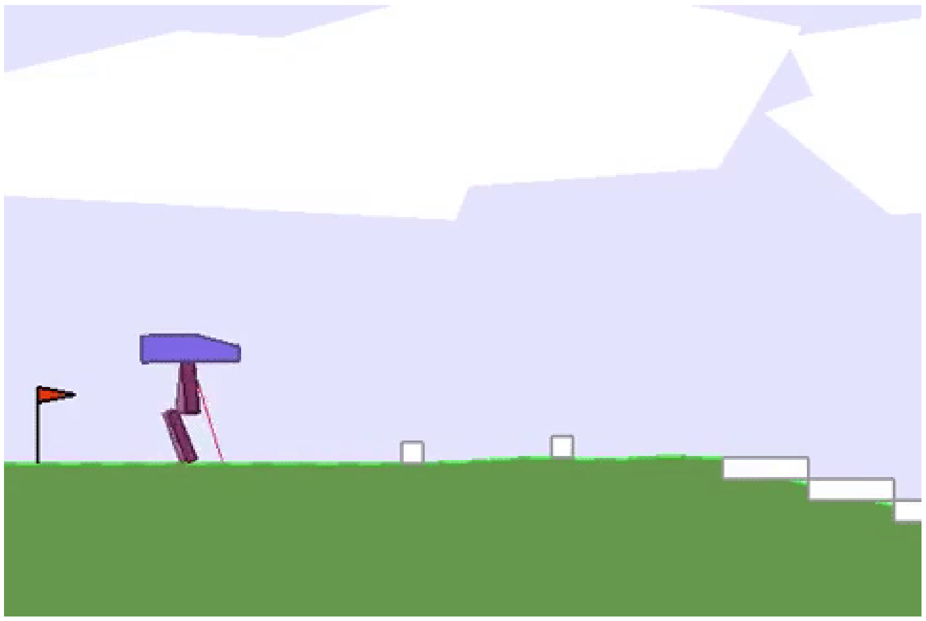

Figure 3: Alt Text

Conclusion

In my mind before I started the project and before I read OpenAI's paper, I had always seen evolutionary algorithms as cute black-box optimzation methods that did not have serious application in training modern machine learning models; however, as we can see in the results section, these algorithms may actually have a place in the traditional reinforcement learning setting. For parameter vectors that aren't incredibly large, like in the millions, evolutionary algorihtms offer an enticing option to train continuous control agents. The ability to

References

1.
2.
3.